PetriLLD Tutorial

James Brusey

PetriLLD Tutorial

by James Brusey

Second Edition

PetriLLD is a graphical development environment that allows the user to construct and test control programs for discrete event systems. The graphical language is a simple form of Petri net with some notational changes that allow it to express sensory input. Once the Petri net is designed and tested, it can be output in a number of different forms, including PLC (programmable logic controller) ladder logic diagrams, and various high-level languages including Java and Visual Basic.

The main features of this tool are the ability to rapidly construct sophisticated control systems that include a large amount of distributed and concurrent behaviour; the ability to compile to both general purpose computer and PLC forms; and the ability to separate the description of the behaviour from the implementation instance.

This tutorial is intended to guide the user through their first use of the tool. It has been written with version 1.1 (build 20060917) in mind, so if you have an earlier version, you might like to start by going through the installation section and installing the latest version.

Table of Contents

Introduction ..... **v**

1. Installing PetriLLD **1**

Installing on Microsoft Windows® 1
Step 1: Ensure that Java 5 is installed 1
Step 2: Install Java if necessary 1
Step 3: Download PetriLLD 1
Step 4: Install PetriLLD 1
Installing on Apple Mac OS/X™ 2
Step 1: Ensure that Java 5 is installed 2
Step 2: Install Java if necessary 2
Step 3: Download PetriLLD 2
Step 4: Install PetriLLD 2
Installing the platform independent version 2
Step 1: Ensure that Java 5 is installed 2
Step 2: Install Java if necessary 3
Step 3: Download PetriLLD 3
Step 4: Execute PetriLLD 3
Compiling from source 3

2. Overview **5**

Entities 5
Tasks 5

3. Tutorial **7**

Getting started 7
Create a new project 7
Creating a net 7
Designing a net 7
Different place types 8
A simple example 9
Testing behaviour 10
Creating instances 10
Compiling the project 11
Downloading your code 12
Toggle button example 13
Drill-press example 14
Adding an instance 16
Introducing project simulation 16
Verification using a model 18
Using high-level language compilers 19
Compiling to Visual Basic 20
Advanced topics 22
Creating modular nets 22
Coordination design patterns 22
Printing and exporting Petri nets 26

A. Reference 27
Elementary Net rules 27
Place types 27
Compiler plug-in example 28
Toolbar 32
Further reading 34

Introduction

PetriLLD is a simple graphical tool that can be used to design PLC programs by building a Petri net that represents the desired behaviour. A screen image is shown in Figure 1.

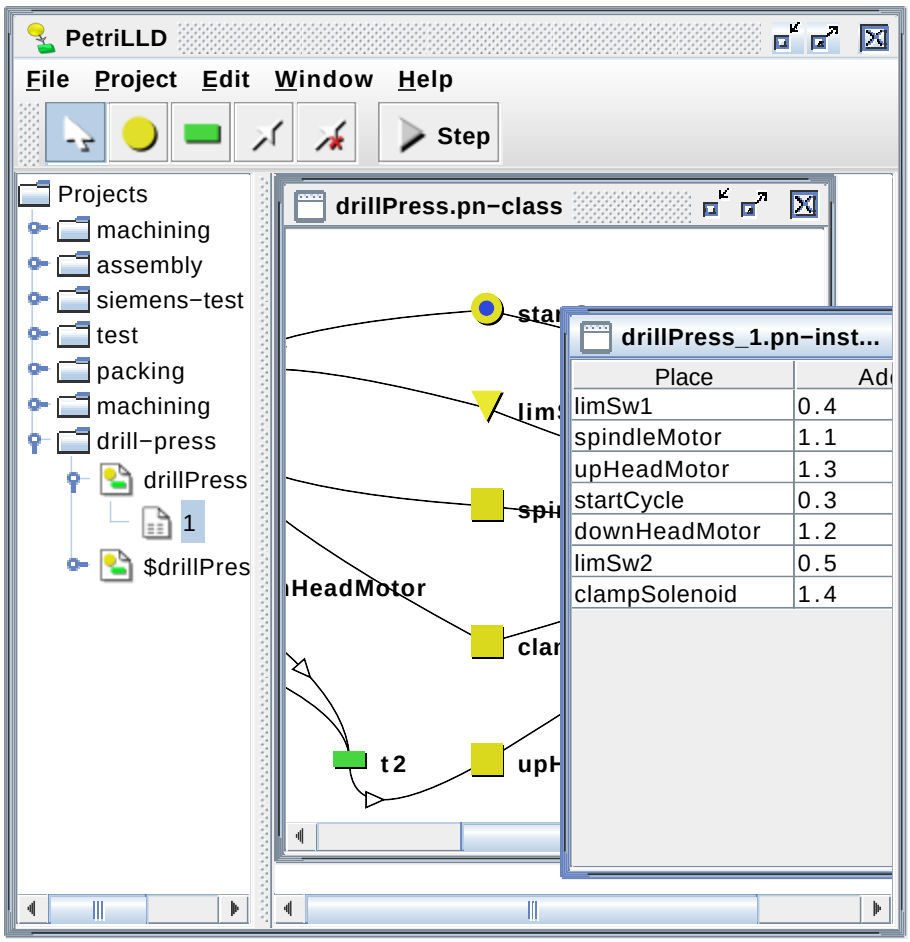

Figure 1. PetriLLD in action

PetriLLD is built upon the Petri net formalism. Specifically, it uses a modified form of a basic sort of net that only allows one token in a place. If you are not familiar with Petri nets, don't worry! Many students have found in the past that just playing around with them and using the simulate tool within PetriLLD is enough to get used to the idea. For a more in-depth treatment, there are also a number of useful resources at the Petri Net World[1] web page. When looking at these resources, keep in mind that nets in PetriLLD include input and output connections to an external environment, which is not usual in ordinary Petri nets. Many of the fundamental ideas, however, are the same.

Petri nets are an excellent model for expressing concurrent behaviour. For this reason, they are useful for modelling the behaviour of discrete event systems such as those in manufacturing plants where there may be many operations occurring simultaneously. PetriLLD was developed with

automatic control systems like those used in manufacturing plants in mind. Nonetheless the tool is intended to be general and may be used for other applications where one needs to express simply some combination of concurrent and sequential behaviour.

PetriLLD was originally devised to produce ladder-logic diagrams for Programmable Logic Controllers (PLCs). The ladder logic diagram language is a graphical language that is analogous to a series of wired connections and switches. In fact, the graphical language is implemented as a series of boolean logical assignment statements. By executing or *scanning* the statements over and over again, the PLC behaves just as if it really contained the wiring and switches shown in the language.

Just like ladder-logic diagrams, PetriLLD turns the Petri net into boolean statements. When loaded into the PLC, it produces the same behaviour as the Petri net, however now it will be affected by actual sensors connected to the PLC and it will turn on and off the PLC's actuators.

In the following tutorial, we will see how this tool can be used to develop control logic. The first step is to obtain a copy of the program and get it installed on the computer that you are using. The installation procedure is described in the next section.

Notes

1. http://www.informatik.uni-hamburg.de/TGI/PetriNets

Chapter 1. Installing PetriLLD

The PetriLLD tool can be run on just about any environment that supports Java™. The only prerequisite is that the Java 5 Runtime Edition must be installed.

Installation consists of two steps:

1. Install Java (if it's not installed already).
2. Download and install PetriLLD.

Installing on Microsoft Windows®

Step 1: Ensure that Java 5 is installed

Go to the **Start** menu and click on **Run...** and then type **cmd** followed by pressing the Enter key. On older versions of Windows, you may need to type **command** instead. This will bring up a DOS prompt. Type the following command to see if you have Java already:

```
java -version
```

You should see something like:

```
java version "1.5.0_06"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_06-b05)
Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode, sharing)
```

This indicates that you have Java version 1.5.0_06. Other versions are acceptable as long as they are greater than 1.5.0. If you see an error message, it probably means that you do not have Java installed.

Step 2: Install Java if necessary

To get Java 5 for Windows, go to java.com[1]. Note that the Java Runtime Environment is all that is required. Once Java 5 is installed, continue to step 3.

Step 3: Download PetriLLD

You can download the latest version of PetriLLD from Sourceforge.net[2]. Unless you want source code, the recommended version for Windows is the one ending "Windows-Installer". Make sure that this downloads as a JAR file. It is not necessary to expand out the JAR file.

Step 4: Install PetriLLD

You should now have a file called "PetriLLD ... Windows-Installer.jar". Executing this file should start the install process. This process is reasonably self-explanatory. If you would like to use the automated installation procedure, refer to the IzPack[3] web-page for documentation on how to use the script file. Note that most users will not need to use this.

Installing on Apple Mac OS/X™

Step 1: Ensure that Java 5 is installed

Start by making sure that you have Java 5 installed. Note that Java 5 may be installed but might not be the default Java. PetriLLD will request Java 5; it is not necessary to change the default.

Step 2: Install Java if necessary

To get Java 5 for Mac OS/X, go to: www.apple.com[4]. Once Java is installed, continue to step 3.

Step 3: Download PetriLLD

You can download the latest version of PetriLLD from Sourceforge.net[5]. Unless you want source code, the recommended version for Mac OS/X is the one ending “MacOSX”. Make sure that this downloads as a DMG file.

Step 4: Install PetriLLD

To install PetriLLD, simply open the disk image (DMG) file and drag the PetriLLD icon into your Applications folder.

Installing the platform independent version

Step 1: Ensure that Java 5 is installed

Open a shell window and type the following command to see if you have Java already:

```
java -version
```

You should see something like:

```
java version "1.5.0_06"
```

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_06-b05)
Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode, sharing)
```

This indicates that you have Java version 1.5.0_06. Other versions are acceptable as long as they are greater than 1.5.0. If you see an error message, it probably means that you do not have Java installed.

Step 2: Install Java if necessary

To get Java 5 for Linux™, go to java.com[6]. Note that the Java Runtime Environment is all that is required. Once Java 5 is installed, continue to step 3.

Step 3: Download PetriLLD

You can download the latest version of PetriLLD from Sourceforge.net[7]. Unless you want source code, the recommended platform-independent version is the one ending "... -bin.jar".

Step 4: Execute PetriLLD

The jar file can be executed directly, as follows:

```
java -jar PetriLLD-1.1-bin.jar
```

Compiling from source

There are a number of prerequisites for compiling from source. First, you need Java 5, as above. Second, you need Ant[8] to perform the build. Third, you will need Batik[9] to support exporting to SVG. Fourth, Ant-Contrib[10] is needed to support the use of conditional statements in the Ant script. To build the Windows Installer, you will need IzPack[11]. Specifically, you need to install the `standalone-compiler.jar` into Ant's `lib` directory. To build the Mac OS/X disk image, you will need JarBundler[12] and the Mac disk image utility (for which you probably need to be running on a Mac!). JarBundler also contains a file called something like `jarbundler-1.9.jar` that should be moved into Ant's `lib`.

Download the PetriLLD file ending in "... -src.jar" and expand it out into a working directory with commands similar to the following:

```
mkdir petrilld
cd petrilld
jar xf ../PetriLLD-1.1-src.jar
```

You may need to modify `build-local.properties` to say where things such as IzPack and Batik are installed. Once this has been done, the next step is to compile the code:

```
ant compile
```

At the time of writing, there are 3 “unchecked” warnings, all of which can be safely ignored.

There are several other targets:

Summary of Ant build targets

run

: Run PetriLLD directly.

compile

: Compile to the build directory,

dist

: Produce a distributable JAR file.

win

: Produce the Windows-Installer JAR file.

macosx

: Produce the Mac OS/X disk image.

Notes

1. http://java.com/java/download/index.jsp
2. https://sourceforge.net/project/showfiles.php?group_id=147649
3. http://www.izforge.org/izpack
4. http://www.apple.com/support/downloads/
5. https://sourceforge.net/project/showfiles.php?group_id=147649
6. http://java.com/java/download/index.jsp
7. https://sourceforge.net/project/showfiles.php?group_id=147649
8. http://ant.apache.org
9. http://xml.apache.org/batik/
10. http://ant-contrib.sourceforge.net
11. http://www.izforge.com/izpack
12. http://jarbundler.sourceforge.net

Chapter 2. Overview

Before starting to use PetriLLD for the first time, it may be useful to understand some of the core concepts used. PetriLLD supports several different entities to allow for a variety of different situations:

- The basic use is to convert a Petri net into low-level ladder logic diagram code that can be downloaded directly into a PLC and used to control a manufacturing cell.
- Several different PLCs may need to be coded for, and in each PLC, several different, and possibly independent devices may need to be controlled.
- Some of those devices may need exactly the same control programs but perhaps with some addresses changed.

Entities

When using the tool, the user manipulates three different types of entity:

- A *project* corresponds to a directory or folder containing all of the code for a single PLC or computer. Within each project there may be a number of *Petri nets*.
- A *net* is the abstract design of a PLC or computer sub-program, that does not specify what sensor or actuator addresses are used. Nonetheless it does specify *which* Petri net places correspond to sensors and actuators. We shall see how this works in the following sections.

  Each net may have multiple *instances*.
- An *instance* specifies what addresses are used when implementing a net for a particular device or set of devices.

These entities are arranged in a hierarchy. A project contains some number of nets, each of which subsequently contains some number of instances. In terms of storing the data onto disc, the project corresponds to a directory, while data associated with nets and instances are stored as files within that directory.

Tasks

PetriLLD provides a simple integrated development environment that supports much of the process of developing robust code. With PetriLLD,

you can:

- Create a new project. You'll need one project for every target PLC or other sort of computer. You can create a new project by right-clicking on the word *Project* in the project window and selecting **New project**. Note that there is an alternative, which is to select *Project* in the left hand window and select the **Project > New project** menu item.
- Design a net to represent the desired behaviour. This means first creating the net and then creating places, transitions and arcs. You may need to change some of the places to be "input" or "output" places so that they can interface with the external environment. This process is discussed in more detail in Chapter 3.
- Test the design under simulation. The simulation tool allows the user to watch how the tokens move from place to place by single-stepping. This allows design faults to be discovered prior to executing on the hardware.
- Connect a net to a set of inputs and outputs by defining an *instance*.
- Compile a set of nets and instances to produce an executable form. Note that various compiler "plug-ins" are provided to allow the design to be compiled to a variety of PLC and computer languages.

Chapter 3. Tutorial

This chapter provides a tutorial introduction to using the PetriLLD tool.

Getting started

The first stage, if you haven't already done so, is to install the software. More information on installation is given in Chapter 1.

Once you have installed it, start the tool. With the program running, you should see a window with a toolbar along the top and the main area split into left and right panes. Note that the divider between the two panes can be dragged to the left or right using the mouse. The pane on the left hand side is the project pane. This shows any projects that you have open. The right hand side is the editor pane. Nets and instances are shown in this pane when they are being edited.

Create a new project

With PetriLLD running, the first step is to create a new project. Right-click on *Projects* to get the context menu for projects and select **New project**. It is also possible to do the same thing by selecting **Project > New project**. Note that if the menu item is greyed out, this means that you need to select *Projects* first.

In the dialog box that comes up, enter:

```
tutorial
```

for the project name. A base folder will already have been selected. If it is not appropriate, you can change it by clicking browse. Note that the base folder must exist and a folder called `tutorial` will be created inside it. Click **Finish** to create the project.

Creating a net

You should now see `tutorial` in the list of projects in the left hand pane. Create a new net called `my-first-net` by right clicking on `tutorial` and selecting **New net**. When this has been done successfully, you will see `my-first-net` as a sub-element of `tutorial`.

Designing a net

Open `my-first-net`. You can do this by either double clicking on `my-first-net` in the project pane or by right-clicking on it and selecting **Open**. When it is open, you'll see a new (blank) window in the editor pane.

You can now create some places and transitions. To do this, select the **Place** or **Transition** tool from the toolbar. (The layout of the toolbar is explained in detail in the Section called *Toolbar* in Appendix A.) Then simply click on some part of the `my-first-net` window to put a place or transition there.

> **Note:** If you run out of room, you can either make the window larger or use the **Select** tool to drag elements off the edge of the screen. In general, you should try and keep your nets simple and able to fit on one screen. Where more complex behaviour is required, split the functionality between different nets (see the Section called *Creating modular nets*).

Once you have both transitions and places in your net, you can join them with arcs using the **Draw arc** tool. To use this tool, first select it from the toolbar, and then click and drag from a place to a transition, or from a transition to a place.

> **Note:** The semantics of Petri nets do not allow arcs from places to places or from transitions to transitions.

Deleting arcs can be performed in one of two ways. You can either delete either the place or transition that the arc connects to, or you can delete just the arc. To delete a place or transition, select it with the select tool and press the delete key. Note that multiple nodes can be selected and then deleted at once. Alternatively, right-click on the place or transition and select the **Delete** option.

To delete an arc without deleting a node that it connects to, use the **Delete arc** tool. To use this tool, first select the tool from the toolbar and then click and drag from the source node to the target node. Note that clicking and dragging in the reverse direction will have no effect (unless there happens to also be an arc from the target to the source).

PetriLLD does not allow the path of the arc to be altered. It is not possible to select arcs or manipulate them. Although this may seem somewhat limiting in terms of improving the look of the Petri net, the current approach seems to have the benefit of simplicity. If the arcs become too messy, consider splitting your net into two or more modules.

Different place types

You can alter the characteristics of a place or transition by right clicking on it and selecting **Toggle input** or **Toggle external**. Places can be turned into input places (represented by a triangle) or external places (represented by a squares). See the Section called *Place types* in Appendix A for

more information about the different sort of places and what they can be used for.

A simple example

Here is a simple example that incorporates some of the features discussed in the previous section. The aim is to draw the net in Figure 3-1. Feel free to skip to the Section called *Testing behaviour* if you have already drawn the net.

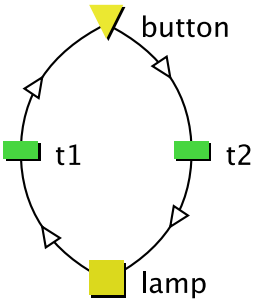

Figure 3-1. A simple net involving a switch and a lamp.

This net produces a simple behaviour. When the switch is turned on, the lamp turns on. When the switch turns off, the lamp turns off.

To produce the above net, first create a new net called `switch_lamp` and open it. Create two places by clicking on the place tool, and then clicking twice on the `switch_lamp` window. Create two transitions in a similar fashion. You may want to refer to the diagram to work out whereabouts to put the transitions relative to the places. Don't worry if you don't get it right as you can move them later.

When the places and transitions are initially created, they receive default names. You can rename them by right-clicking on them and selecting **Rename**. Rename one of the places to **`Switch`** and the other to **`Lamp`**. Leave the transitions with the default names.

Now connect the places and transitions with arcs as shown in the diagram. Start by selecting the **Draw arc** tool. Note that the direction of the arc is important. For example, to make an arc from `Switch` to `t2` move the cursor over `Switch`, press down the left mouse button, drag the cursor to `t2`, and then release the button. As you drag the mouse, you'll see a red line indicating where the arc will be created.

Once you have completed the arcs, all that is required is to set the place types for the switch and lamp. This is done by right clicking on `Switch` and selecting **Toggle input**. Similarly, right-click on `Lamp` and select the **Toggle external** menu item.

> **Note:** Square nodes are “external” and not “output” because they can also be used to receive messages from other nets or even other computers. See the Section called *Shared memory coordination pattern* for further information

At this point, you may notice that the arcs for one of the transitions seem to have a convoluted shape. This is due to the arcs curving so that they come in to the top of the transition and leave from the bottom. It is possible to adjust this behaviour by right-clicking on one of the transitions and selecting **Flip transition**.

Testing behaviour

Once you are happy with the net, you can test its behaviour by clicking on the **Step** button. If there are no transitions that can fire, you will see no change in the state of the net. To allow one of the transitions to fire, right-click on `Switch` and select **Toggle mark**. You will then see a mark (or token) appear in the `Switch` place. Pressing **Step** will then cause the lamp to turn on. Note that since `Switch` is an input, it will not change its state until you manually adjust it with the **Toggle mark** option.

With the `Switch` turned on, `Lamp` becomes marked and stays marked no matter how many times the **Step** button is pressed. If `Switch` is toggled to an unmarked state, subsequently pressing **Step** will cause `Lamp` to turn off. Note that in both cases, the firing of a transition does not affect the state of `Switch`.

> **Note:** In general, arcs that flow *from* an input place require that place to be *turned on* (marked) for the associated transition to fire. Conversely arcs that flow *to* an input place require that the place be *turned off* (unmarked) for the associated transition to fire.

As a small exercise, see if you can make the lamp turn on only when the switch is off (i.e. reverse the polarity).

Creating instances

At any stage after creating the net, one or more *instances* can be created. An instance is required to tie input places and external places to input and output (or shared memory) addresses, respectively. To create a new instance, right-click on the `switch_lamp` net and select **New instance**. This will prompt for an instance name. Enter

```
1
```

as the name.

> **Note:** Instance names are appended to the net name when generating the filename. Also, for some compiler plug-ins, variable names are generated by concatenating together the net name, the instance name and the place or transition name. For this reason, it is usually a good idea to avoid having long net or instance names, as it can make the generated code difficult to read.

An instance is essentially a mapping of input and external places to addresses. The format of an address will depend on the compiler and the target PLC or computer. For example, for an OMRON PLC, address consist of a word address followed by a "." followed by a bit address. For example **`100.02`** refers to the third least significant bit in word 100. (It's the third bit, since the least significant bit is bit 0.) Assuming that you are using an OMRON PLC, enter

```
0.00
```

for place `Switch` and

```
4.00
```

for place `Lamp`.

Compiling the project

Once at least one instance has been created, it is possible to compile the project.

> **Note:** It is not possible to compile nets or instances independently from a project. Only projects can be compiled.

To compile a project, right click on the `tutorial` project in the project pane and select **Compile all**.

A dialog box is shown that allows the compiled output format to be selected. These compiler "plug-ins" are discussed in more detail in the Section called *Compiler plug-in example* in Appendix A. Following this, it is possible to select where to write the compiled output format. This file can then be loaded into the utility or compiler.

For example, the diagram in Figure 3-2 is the ladder logic that results from compiling to OMRON CX-Programmer (shown without optimising unnecessary transitions).

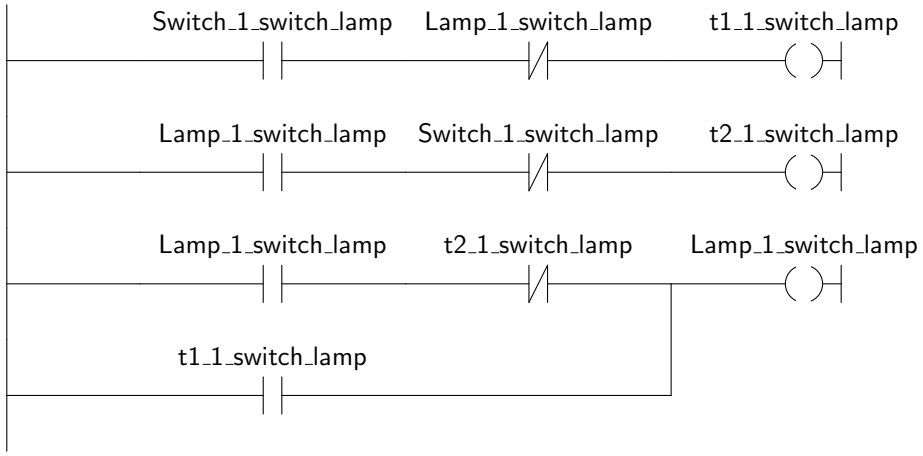

Figure 3-2. Ladder-logic diagram produced by compiling `switch_lamp`.

With optimisation, the code in Figure 3-2 simplifies to that in Figure 3-3. PetriLLD's optimisation algorithm works by first detecting particular cases where transitions can be eliminated by substituting their update expression wherever they occur. Unfortunately, there are only a few cases where this may be performed without causing a large increase in the complexity of the resulting code (the case shown here being an example). Therefore, PetriLLD limits the situations where it applies this optimisation approach.

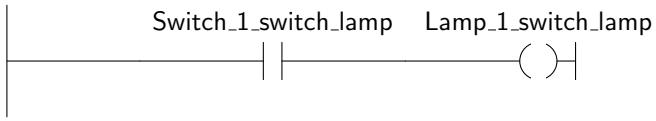

Figure 3-3. Simplified version of the previous ladder logic.

Downloading your code

The compiled code needs to be downloaded to the PLC for it to take effect. The exact method varies depending on download software. Note that PetriLLD doesn't take care of this part for you—the download is a manual process.

The process to download to an OMRON CS1 PLC involves opening the file produced by PetriLLD with OMRON CX-Programmer, or CX-One depending on which software release you are using. Note that the filename ends in `.cxt`, and it is necessary to tell CX-Programmer to look for this type of file. Once the file has been opened, it is necessary to alter the network settings to suit the target PLC. Once this has been done, download the program to the PLC. Note that PetriLLD does not code an I/O table, and it is thus unnecessary (and unwise) to try to download it. Therefore, ensure that the download options only have *Program* ticked.

At this stage, you may discover some bug in your code. Tools such as CX-Programmer can help you identify the cause of the problem. Remember,

however, that it is not possible to change the ladder logic and then automatically translate that back into a Petri net. Therefore, if you do decide to change the code, it is recommended that you change the Petri net and recompile.

Toggle button example

The following is a slightly more complicated example that involves a lamp that is toggled on and off. That is, when the button is pressed and released, the lamp turns on if it is currently off, or off if it is currently on.

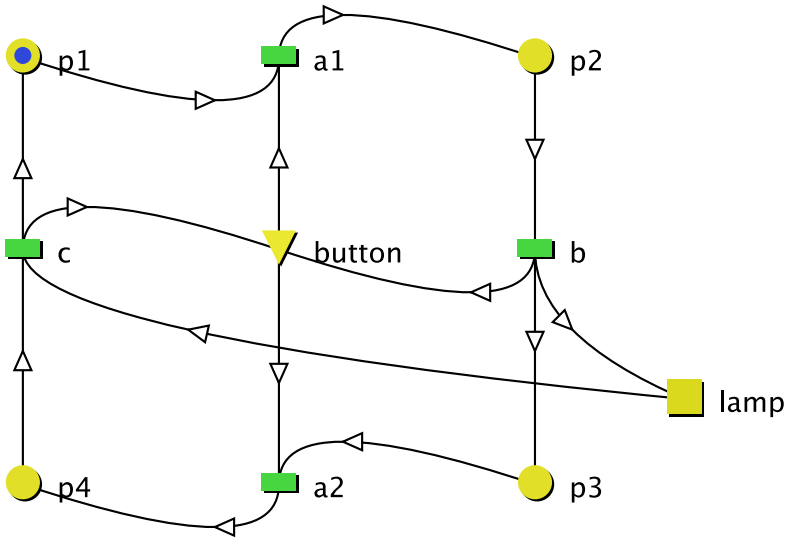

Figure 3-4. Toggle button Petri net

Figure 3-4 shows the solution to this problem. The system moves through four main states, starting in `p1`, moving through to p4 and then back to `p1` again. To get an idea of how this net works, consider each transition. Transition `a1` occurs when the net is in an initial state, and the button is depressed. When the button is released (turned off), transition `b` will fire. This also turns the lamp on. Transition `a2` fires when the button is pressed again, while `c` occurs when the button is released.

Note that in Figure 3-4, `p1` is initially marked. To set the initial marking for any net, use **Toggle mark** to put the net into the desired state, then select **Edit > Set initial marking**.

The behaviour of the above net can also be produced with a net with slightly fewer places (this is left as an exercise to the reader). Note, however, that it is not possible to bypass the step of looking for the button turning on (`a1` and `a2`). If those transitions were deleted from the loop, the system would just flash the lamp on and off (assuming the button was off or released).

Note: Inputs such as buttons tend to need arcs going both out and in. If you find yourself designing a net with arcs only coming out of inputs, check to see that this is not a potential problem.

Drill-press example

So far, the examples looked at bear little resemblance to realistic automation problems. In the following example, automation for a drill press, as shown in Figure 3-5, is developed.

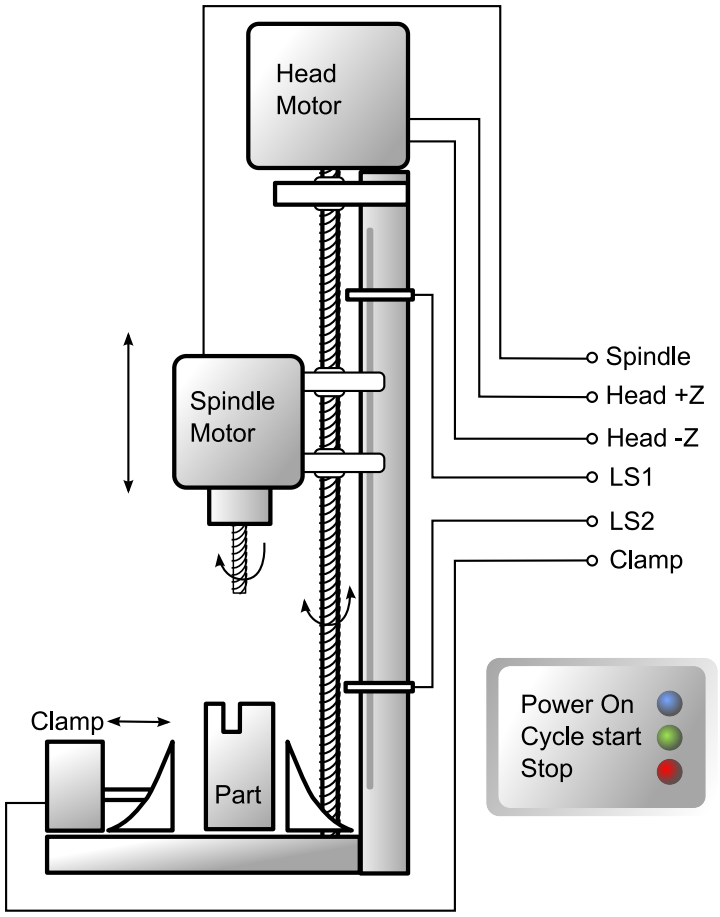

Figure 3-5. Schematic of a drill press

The drill press consists of some sensors (limit switches LS1 and LS2), some momentary push buttons (power on, start cycle, stop), some lights (power, cycle start), an actuator (fixturing clamp), and some motors (+Z, -Z, and spindle motors). For the purpose of this example, the control code will meet a simple set of requirements:

i. The drill head starts in the upper most position, with the upper limit switch `LS1` turned on.

ii. When the start cycle button is pressed, the clamp should activate, the drill should start moving down and the spindle motor should turn on.

iii. When the lower limit switch `LS2` is reached (and thus turns on) the downward (-Z) drill head motor should turn off, the upward (+Z) motor should turn on.

iv. Finally, when the upper limit switch is reached, the clamp should be released, the spindle motor turned off, and the upward motor turned off.

The process of translating these requirements into a net can be broken into several steps:

1. Create places for each input (sensors and buttons) and for each output (motors and solenoids).
2. Modify the places to match their type (make sensors and buttons input-only, and actuators external).
3. Create a transition to correspond to each event. For example, you'll need a transition for starting the process, one for when the drill reaches the lower limit switch, and one for when it has returned to the upper limit switch.
4. Join transitions and places with arcs. Generally, you can view the input arcs from X, Y, Z, to a transition as saying X and Y and Z must be true. Nevertheless, keep in mind that, for non-input places, the transition also draws the tokens from X, Y, and Z.
5. Add any internal state places as necessary.
6. Refine the net to remove any unnecessary items.

Figure 3-6 shows a possible solution to the drill press control problem.

Note that there are arcs only coming out of (say) `startCycle`. As noted previously, this may indicate a potential fault in the design. What potential problem might occur in this case?

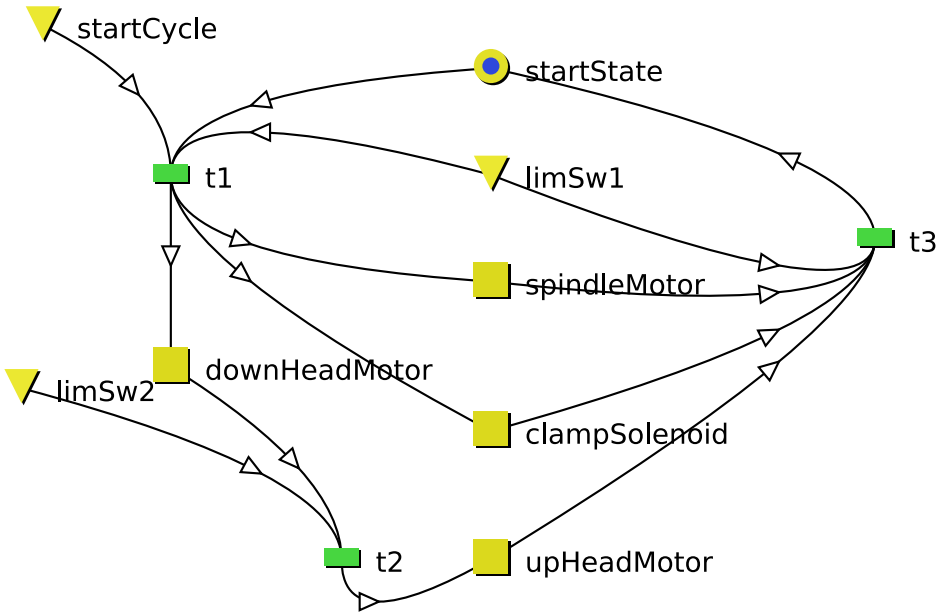

Figure 3-6. Petri net controller for the drill press

Adding an instance

Whether you wish to control three drill presses or thirty drill presses, you can use the same net over and over again. If you need to control different devices with different PLCs, then manually copy the `.pn-class` file to other project directories. It is expected that future versions of PetriLLD will support libraries of nets. To allow several different drill-presses to be controlled by a single PLC, simply create a separate instance for each one.

The instance contains a mapping between external and input places to PLC addresses. An example mapping is shown in Table 3-1.

Table 3-1. Drill press instance example

Place	Address
clampSolenoid	1.4
downHeadMotor	1.2
limSw1	0.4
limSw2	0.5
spindleMotor	1.1
startCycle	0.3
upHeadMotor	1.3

Adding an instance is required if you want to include the drill press net in the compiled output, or in the project simulation (see the next section). Nets without instances are ignored when compiling or using the project simulator.

Introducing project simulation

Once the net and instance have been created, you may wish to verify that it works correctly before testing it with real equipment. One way to do this is to use the **Step** tool to examine the behaviour. A more sophisticated simulation tool is provided in the form of a project simulator. You can access this feature by clicking on the project that you want to simulate and then selecting **Project > Simulate project**.

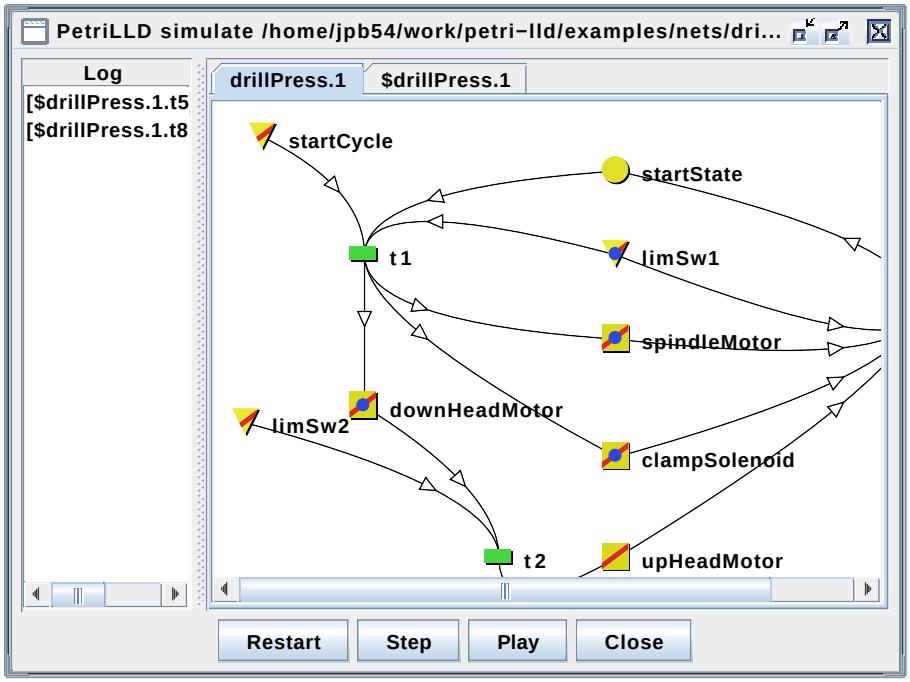

Figure 3-7. Using the project simulator

The project simulation tool, shown in Figure 3-7 allows simulation of not just a single net, but a whole project, potentially including many nets and instances. In some cases, place addresses will be shared between several instances, and this is identified in the simulator by a red slash through the place.

> **Tip:** Moving the mouse pointer over any place will reveal the address. If the place is shared, a list of other places sharing the address will be shown in square brackets.

The log window on the left hand side shows a list of transitions in the order that they occurred. It is possible for two transitions to occur during the same scan cycle, and this is shown by having all simultaneously occurring transitions appear on the same line in the log. Note that the log is cleared when **Restart** is pressed.

The project simulator can either be run in single-step mode by clicking **Step**, or be run in continuous play mode by clicking **Play**. While continuous mode runs, the **Play** button changes to a **Stop** button. Pressing

Close closes the simulation window. Note that it is necessary to stop the simulation before closing the window.

At any time, it is possible to manually change the state of places by clicking on them. This toggles their state from marked to unmarked. So that you know what you've done, this is also recorded in the log.

The project simulator supports simulation of time delays on transitions whereas the single-step simulator available when editing nets does not. In continuous play mode, the simulator behaves as though the scan cycle time is around 100 milliseconds.

Verification using a model

The simulator allows basic testing that may help find bugs in control logic. More sophisticated verification can be performed by developing a net to model the uncontrolled behaviour of the device being controlled, and using this to automatically test the control logic.

> **Note:** To support the use of "simulator-only" nets, the compiler excludes any nets that have a name starting with `$`. Thus, for a controller net called `drill-press`, the associated model net might be called `$drill-press`.

A good place to start with modelling the behaviour of a device is to create input places where the controller net has output (external) places and external places where the controller has inputs. It is a good idea to use a consistent naming convention to make it easy to ensure that equivalent places in the model and controller are wired to the same address.

The next step is to model the possible states of the device or system being controlled. This may or may not correspond to the state of sensors. In many cases, the state of a device cannot be unambiguously identified by looking at the state of the sensors associated with it. To model the drill press described in the Section called *Drill-press example*, you might start by modelling the position of the drill head. For example, the drill head can be considered to be in one of three possible states: fully up, fully down, or somewhere in between. Fortunately, the two limit switches correspond to the fully up and fully down states, so it is only necessary to represent the state of being "in the middle".

Next, the movement between states must be represented. Each possible way of going from one state to another must be represented by a transition. For example, going from `middle` to `up` will have a transition that will also have an arc from the upward motor. The final result for the drill press should look something like Figure 3-8.

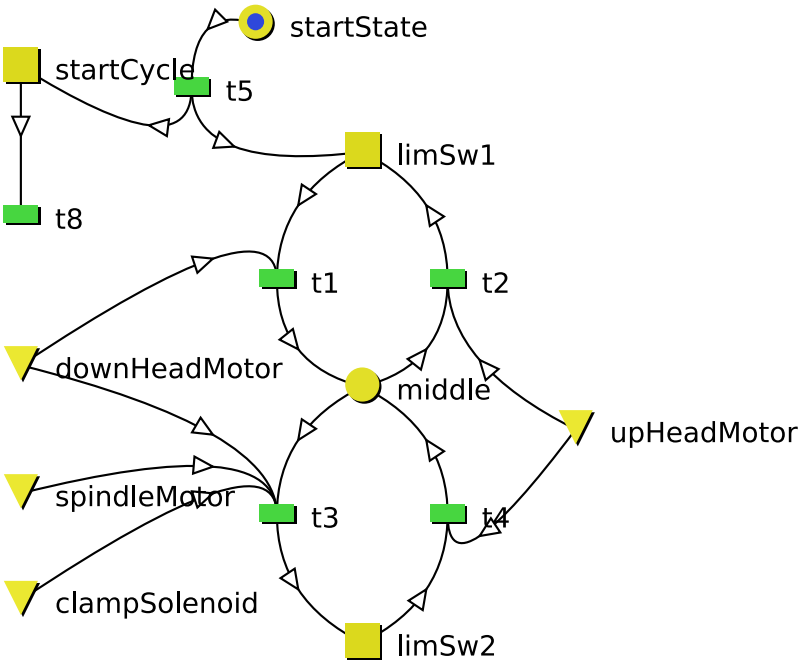

Figure 3-8. A model of the drill press: `$drill-press`

There are several other aspects about the design of Figure 3-8 that are worth noting. First, `startCycle` is turned off immediately by `t8`. Setting a time delay for `t8` will cause `startCycle` to stay on for some period.

> **Note:** The use of time delays in the controller is generally a sign of "open-loop" control. The time factor is used as a proxy for being able to directly sense a change in the environment. This is sometimes necessary but should be avoided if possible. Using time delays in the model, on the other hand, is generally less problematic.

Using high-level language compilers

Rather than compile to a PLC language, it is also possible to compile to a general purpose programming language. Currently compiler plug-ins for two languages are supported: Visual Basic and Java. The Visual Basic plug-in is described in detail below, but both are quite similar.

There are several reasons for compiling to a high-level language. For example, you may want to just simulate the behaviour of a PLC. The main reason for this feature being developed, however, was to provide a mechanism for coding complex logic that connects and interacts with Petri nets on the PLC. When faced with this type of problem, software developers usually code a linear interaction where the high-level program asks the PLC to do something and then waits for the PLC to complete. In order to perform several such interactions simultaneously, it is necessary to use multiple threads. This tends to create highly complex programs that are

difficult to debug. An alternative is to formulate the program as a Petri net, and that is the approach described below.

In the next section, we look at what code is produced. Following that, we examine how to use it in the context of PLC interaction.

Compiling to Visual Basic

To produce Visual Basic (VB) from your PetriLLD project, click on the project and select **Project > Compile project**. Note that only nets with instances will be compiled. Select *Visual Basic Infix* as the compilation format. The VB class name is determined based on the file name that you choose. So, for example, setting the filename to `Foo.vb` will produce a class called `Foo`.

A natural application for nets in VB is to use them to interact with nets running on a PLC. To make this happen, it is necessary to add some additional code to that produced by the VB plug-in in order to couple it to the PLC. This additional code needs to

i. detect when certain PLC memory locations change and to update the associated VB object, and,

ii. detect when the net running in VB changes an object and to update the associated PLC memory location.

> **Warning**
>
> Do not edit the file generated by PetriLLD as any edits will be removed by the next compile. All customisation should be done by sub-classing BoolVar and / or writing code that occurs before and after the call to DoStep.

To understand how to do this, it is necessary to examine the generated VB code in more detail.

The structure of the class produced is as follows.

```
Public Class DrillPress
 Inherits AbstractNet (1)
 Private spindleMotor_1_drillPress as BoolVar (2)
 ...
 Public Sub New(ByRef dict as System.Collections.Hashtable) (3)
 spindleMotor_1_drillPress = lookup(dict, "1.1") (4)
 ...
 End Sub
 Public Function DoStep () as Boolean (5)
 ...
```

Figure 3-9. Generated Visual Basic code

(1) The class inherits from AbstractNet, which is found in `AbstractNet.vb` in `examples/vb` folder, distributed with PetriLLD.

(2) The DrillPress class interfaces with the external boolean variables by updating a BoolVar object. See `BoolVar.vb` in the examples folder.

(3) When creating a new DrillPress instance, you need to pass in a dictionary in the form of a hash table. This should contain a mapping between addresses and BoolVar objects. See the example in the `examples/nets/drill-press` folder.

(4) This section of the code can be used to determine which addresses need to be mapped in the dictionary that you pass to the constructor.

(5) `DoStep` should be called to perform a single "scan cycle". If you want to run the net continuously in the background, use a timer to call `DoStep` periodically (perhaps, every 100 milliseconds). Note that if you are interacting with a PLC, you also need to

i. update any BoolVar objects based on the PLC prior to calling `DoStep`

ii. update the PLC memory when any BoolVar objects are changed after calling DoStep.

`DoStep` returns `True` if any transition fired. In this case, the net may still be live.

Note that there are several additional files that may need to be included into your Visual Basic project to allow the generated code to compile (and run!). These are all in the `examples/vb` folder in the PetriLLD distribution.

Performance considerations

The performance of the PC to PLC interaction can be improved in several ways:

1. If your PLC supports it, use PLC event monitoring to trigger updates to BoolVar objects from the PLC. Optimally, the PLC should cause a VB method to be executed when any one of a small set of memory areas change. This small set should include all BoolVar objects. As a further improvement, call `DoStep` as soon as a BoolVar object is changed by the PLC.
2. Derive a sub-class of BoolVar that updates the PLC only when a change occurs. See `MonitorVar.vb` for an example of how to detect when the change occurs.
3. Use bit-wise updates and read accesses to the PLC if possible. Note that if you must update a word of memory at a time, be careful that this does not potentially overwrite an update being performed by the PLC.

4. Call `DoStep` repeatedly (without delay) if it returns `True`. It may be necessary to limit the maximum number of repeated calls to ensure that the user interface remains responsive. Note that it is not usual for a controller net to be constantly live; if it is, their may be a fault in its design.

Advanced topics

Creating modular nets

In any large system, modularity is an important tool for reducing complexity. PetriLLD has some basic support for creating modular nets by allowing them to share addresses. In the future, it is expected to provide a function block-like editor that allows the user to graphically draw the connections between nets. For the moment, it is necessary to make the connections by assigning the same address to places in two different nets.

There are some general rules that you should follow when connecting nets together with shared places.

i. If the nets run on different physical computers or PLCs, make sure that there are only arcs going into any particular (non-input) shared place in one net and out of that place in the other. This form can be used for a token passing structure (see the Section called *Shared memory coordination pattern*). This rule does not apply to input places. Note that only two computers should share any particular (non-input) place.

   In the case where two separate nets on different PLCs or computers need to share a resource, do not directly share the place associated with the resource. One net or other should own the resource and to provide an interface allowing the other to request it.

ii. If two or more nets run on the same computer or PLC, PetriLLD guards against conflicting transitions firing simultaneously and it is thus possible to have arcs going into and out of a place in all nets that share the place.

Coordination design patterns

In this section, two patterns for coordination between two separate computers (or PLCs, robots or other devices) are described. The first approach

is simpler but requires shared memory. That is, it requires memory that can be updated by both computers (or PLCs or whatever). This is referred to as the shared memory coordination pattern. The second approach is more complex but can be used where no memory is shared. Instead of shared memory, it requires an input and an output connection.

Shared memory coordination pattern

The shared memory coordination pattern can be thought of as a token passing approach. In essence, a token (or place marking) passes from the client (requester) to the server. The client-side Petri net is shown in Figure 3-10 while the server-side is shown in Figure 3-11. Note that the external place `Go` is mapped to the same address in both the client and server. Similarly with the `Done` place. That means that when `Go` becomes marked in one net, it also becomes marked in the other.

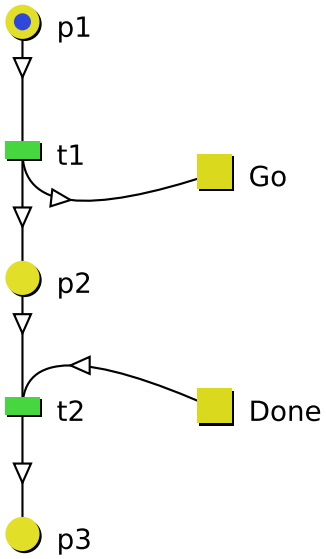

Figure 3-10. Shared place coordination: client-side

As shown in Figure 3-10 the client initiates the process by passing a token into `Go`. The client then waits for `Done` before continuing. Note that although it may seem to be the case that `p2` could be done away with, this place provides for the possibility that other nets make use of the same `Go` and / or `Done` signals. This design prevents this net from grabbing a `Done` token meant for someone else.

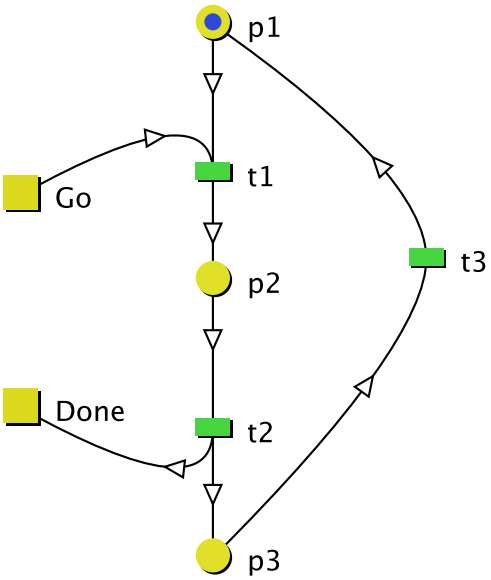

Figure 3-11. Shared place coordination: server-side

As can be seen from the net in Figure 3-11, the server responds to `Go` by going into `p2`. The transition `t1` should also have arcs (not shown in this figure) to start whatever processing is required of the server. Similarly, `t2` should receive arcs from this process indicating its termination.

The above design pattern can be reshaped in a number of ways. The key ideas are (a) to use a shared place (go) to transmit a token and receive it back using a separate shared place (done), and (b) to keep track of waiting for the receipt of the token back.

Coordination with a wired connection

When shared memory is not available, it may instead be possible to wire the input of one device to the output of the other and vice versa. The following solution makes use of two such wires, one expressing "Go" from the client to the server and the other expressing "Busy" from the server to the client. The pattern is expressed in Figure 3-12 and Figure 3-13. The idea is for one side to set a signal high until it can be sure that the other side has received that signal. It is important that neither side restarts the process until both have finished.

Note that, in comparison with the previous pattern, an output place in one net (such as `Go` in the client) is connected to an input place in the other.

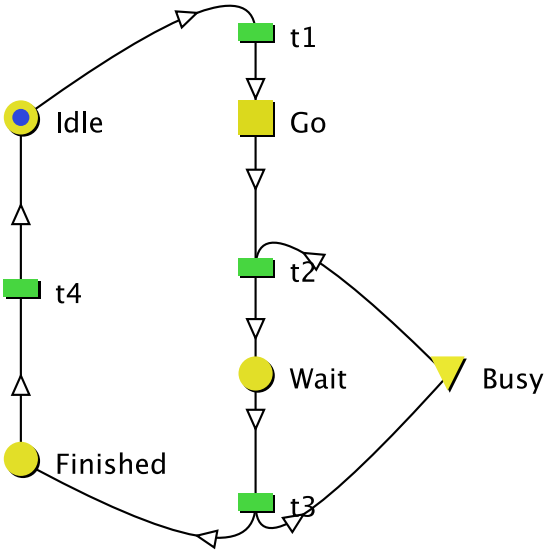

Figure 3-12. Wired coordination pattern: client-side

The process starts with the client setting the `Go` signal. Note that `Busy` should not be set high initially. For this reason, this signalling connection cannot be shared by different nets. When `Busy` becomes high, the client responds by removing the token from `Go`. It stays in `Wait` until `Busy` goes low, thus indicating the end of the process.

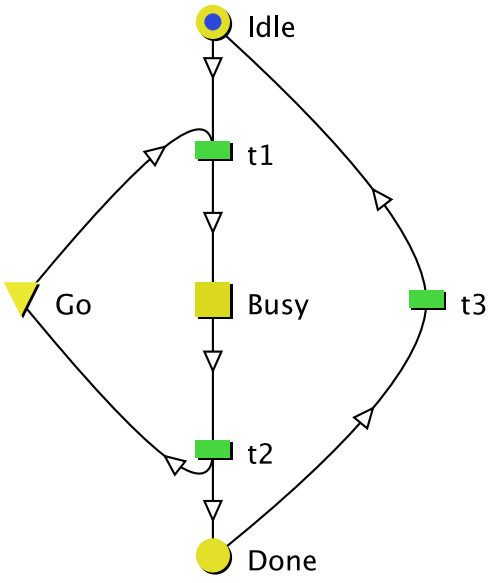

Figure 3-13. Wired connection pattern: server-side

The server side of the wired coordination pattern, as shown in Figure 3-13, is somewhat similar to the client. Transition `t1` should have an outflowing arc (not shown in the figure) that initiates the process being requested. Similarly, `t2` should have an inflowing arc from the completion of that process. Note that `t2` cannot fire until `Go` has been set low by the client. This protects against the situation where the client is much slower than the server and does not get around to updating `Go` in

time. Without this check, the process may be initiated several times in the server when the client only intend for a single initiation.

Printing and exporting Petri nets

Once your Petri net has been completed, you may wish to print the net design, or export it to a graphical format. PetriLLD provides some basic facilities for performing both these tasks.

To print a net, use **File > Print** with the net open and selected. It is also possible to print instances in this way.

To export a net as a graphical image, select the net and then select **Project > Export to SVG**. This produces a Scalable Vector Graphic.

If your favourite word processor does not support directly importing SVG files, try using ImageMagick[1] to convert the file into a more suitable form. For example, with ImageMagick installed

```
convert -size 200x200 test.svg test.wmf
```

converts `test.svg` to Windows Metafile (WMF) format.

Notes

1. www.imagemagick.org

Appendix A. Reference

This appendix is a general reference for the PetriLLD tool.

Elementary Net rules

PetriLLD is based on an extension of a mathematical model known as an "Elementary Net". There are a number of basic rules that govern when transitions are enabled, and what happens when they fire. Note that these rules are slightly adjusted from the standard rules for elementary nets to allow for the inclusion of *input places* in the model.

The rules are:

1. An individual transition is enabled if all of its preconditions are marked and all of its postconditions are unmarked. By *precondition*, we mean a place connected to the transition via an arc *leading to* the transition. Similarly a postcondition is a place with an arc *leading from* the transition.
2. A transition can only fire when it is enabled. When a transition fires, all of its preconditions, with the exception of any *input places*, become unmarked and all of its postconditions, with the exception of any input places, become marked. *Input places* are a subset of the set of places that are never affected by transition firing.
3. A set of enabled transitions can fire simultaneously as long as they do not share any preconditions or postconditions.

Due to the ordering of evaluation of rungs within a PLC, one transition takes priority over another transition, if it occurs earlier in the evaluation sequence. This priority ordering also acts as an arbitrator in the case of conflict.

Place types

Places can be of one of three types:

Ordinary places

> Ordinary places are represented by circles and are not directly affected by external events. Nor do they directly control actuators or other external outputs. Ordinary places are assigned addresses automatically and the actual address location may change from compile to compile. Ordinary places have two states, marked and unmarked.

Input places

Input places are represented by triangles and are typically only connected to inputs. For example, an input place might represent the state of a binary proximity sensor. When the sensor value is high (corresponding to a bit value of 1), then the input place is marked. Input places have the characteristic that they are not affected by the firing of transitions.

Input places have manually assigned addresses and therefore every instance of a net must provide an actual address for each input place in the net.

External places

External places are represented by squares and are typically connected to outputs. They may also represent communication areas between two nets. If the memory location for the external place is shared between two different devices, external places can be used to communicate with external computers.

External places, as with input places, have manually assigned addresses and each instance must provide the actual address.

Compiler plug-in example

Writing a plug-in to generate other sorts of code is quite easy. A small example that generates Lisp-like code is shown in Figure A-1.

```
package uk.ac.cam.eng.pnlld.compile;
import uk.ac.cam.eng.pnlld.*;

/**
 * A simple Lisp code generator.
 * @author James Brusey
 */
public class MyLispGen extends PeInfixCompiler (1)
{
    public PeCompileFilter getFileFilter() { (2)
        return new PeCompileFilter() {
            public static final String EXT = ".lisp";
            private static final String DESC = "Lisp (*.lisp)";
            public String getDescription() {
                return DESC;
            }
            public String getExtension() {
                return EXT;
            }
        };
```

```
}

private ExprFactory expr_factory = new LSExprFactory(); (3)

class LSTrue extends True {
    public LSTrue() {
        super();
    }
    public String print() {
        return "T";
    }
}
class LSFalse extends False {
    public LSFalse() {
        super();
    }
    public String print() {
        return "F";
    }
}
class LSVar extends Var {
    public LSVar(CompNode n) {
        super(n);
    }
    public String print() {
        return name();
    }
}

class LSLet extends Let {
    public LSLet(Var v, Expr e) {
        super(v,e);
    }
    public String print() {
        return "(setq " + var.print() + " " + expr.print() + ")";
    }
}

class LSAnd extends And {
    public LSAnd(Expr a, Expr b)
    {
        super(a, b);
    }
    public String print() { (4)
        return "(and " + a.print() + " " + b.print() + ")";
    }
}

class LSOr extends Or {
    public LSOr(Expr a, Expr b) {
        super(a, b);
```

```
        }
        public String print() {
            return "(or " + a.print() + " " + b.print() + ")";
        }
    }

    class LSNot extends Not {
        public LSNot(Expr a) {
            super(a);
        }
        public String print() {
            return "(not " + a.print() + ")";
        }
    }

    class LSStartTimer extends StartTimer {
        public LSStartTimer(Expr e, int timer, int delay) {
            super(e, timer, delay);
        }
        public String print() {
            return "(time-delay " + a.print() + " " + timer + " " + delay + ")";
        }
    }

    class LSIsComplete extends IsComplete {
        public LSIsComplete(int timer) {
            super(timer);
        }
        public String print() {
            return "(is-complete " + timer + ")";
        }
    }

    class LSExprFactory extends ExprFactory {
        public True makeTrue() {
            return new LSTrue();
        }
        public False makeFalse() {
            return new LSFalse();
        }
        public Let makeLet(Var a, Expr b){
            return new LSLet(a, b);
        }
        public Var makeVar(CompNode n){
            return new LSVar(n);
         }
        public And makeAnd(Expr a, Expr b){
            return new LSAnd(a, b);
         }
        public Not makeNot(Expr b){
            return new LSNot(b);
```

```
        }
        public Or makeOr(Expr a, Expr b){
            return new LSOr(a, b);
         }
        public IsComplete makeIsComplete(int timer){
            return new LSIsComplete(timer);
         }
        public StartTimer makeStartTimer(Expr e, int timer, int delay){
            return new LSStartTimer(e, timer, delay);
        }
    }

    protected void writeFileHeader(Object [] props,
         PeProject project, CompiledNet net)
    {
        out.println("(defun dostep ()"); (5)
        out.println("  (let (");
        writeAddressList(net); (6)
        out.println("       )");
    }

    protected void writeAddress(String var_name, AbstractAddress address,
          boolean is_automatic, boolean is_transition) {
        out.println("   (" + var_name + " F)"); (7)
    }

    protected void writeFileTrailer(Object [] props, CompiledNet net)
    {
        out.println(")"); (8)
    }

    protected void writeStatement(Expr e)
    {
        if (e == null) return;
        for (String s : e.print().split("\n"))
            out.println("     " + s); (9)
    }

    protected ExprFactory getExprFactory() {
        return expr_factory;
    }

    public AddressFactory getAddressFactory() {
        return StringAddress.getFactory(); (10)
    }

    public String toString() { return "My Lisp Example (alpha)"; }(11)
}
```

Figure A-1. Lisp Generator

(1) The class should extend PeInfixCompiler. This generates statements based on inner classes from PeExpr.

(2) The file filter defines acceptable filename extensions and a descriptive name for the output file format.

(3) An ExprFactory instance is required that can create instances of the various PeExpr inner classes.

(4) This is an example of one of the PeExpr inner classes. Here LSAnd defines how to write a binary "and" as a Lisp expression.

(5) The `writeFileHeader` method can be used to produce the first part of the file, possibly including variable declarations.

(6) Variable declarations are produced by calling `writeAddressList`. This will call `writeAddress` for each address entry.

(7) The `writeAddress` method should define how to write out variable declarations.

(8) The `writeFileTrailer` method writes everything after the main body. It is possible to call `writeAddressList` here rather than in `writeFileHeader`.

(9) The `writeStatement` method writes out each statement. A statement usually corresponds to a single rung and generally corresponds to either a "Let" statement or a "Start timer" instruction.

(10) The address factory is used to define the format of addresses for this PLC or language.

(11) The string produced here is used when presenting the user with a list of possible output formats for the compilation process.

To install a new plug-in, edit the code for `PeCompileStrategy` to add your compiler to the list of possible ones.

Toolbar

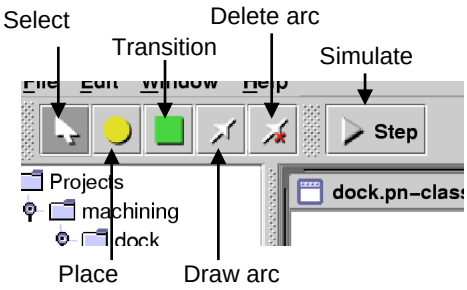

Figure A-2. Toolbar

The toolbar is used when editing Petri nets (i.e. pn-class files). The tools are:

Select tool

Select and move places or transitions.

Place tool

Create a new place.

Transition tool

Create a new transition.

Draw arc tool

Create arcs between transitions and places (or vice versa). Note that you need to press and hold down the mouse button when over the source node and release when over the target node.

Delete arc tool

Delete arcs. This tool is used in the same way as the draw arc tool, hold down the mouse and drag from the source to target.

Simulate tool

Perform a single simulation step. This affects the status of marks according to Petri net semantics and can be used to test a net design.

Further reading

Thomas Boucher, *Computer Automation in Manufacturing*, Chapman and Hall, 1996.

Wolfgang Reisig and Grzegorz Rozenberg, *Lectures on Petri Nets I: Basic Models*, Springer Verlag, 1998.

James Brusey and Duncan McFarlane, "Designing Communication Protocols for Holonic Control Devices using Elementary Nets", *Proc. 2nd Intl. Conf. on Application of Holonic and Multi-Agent Systems (HoloMAS 2005)*, Springer Verlag, 2005.

James Brusey and Duncan McFarlane, "Non-autonomous elementary net systems and their application to Programmable Logic Control", *IEEE Systems, Man, and Cybernetics, Part A*, IEEE Press, (to appear).

Rene David and Hassane Alla, *Discrete, Continuous and Hybrid Petri Nets*, Springer Verlag, 2004, 3-540-22480-7.

Christos Cassandras and Stéphane Lafortune, *Introduction to Discrete Event Systems*, Springer, 1999, 0-7923-8609-4.